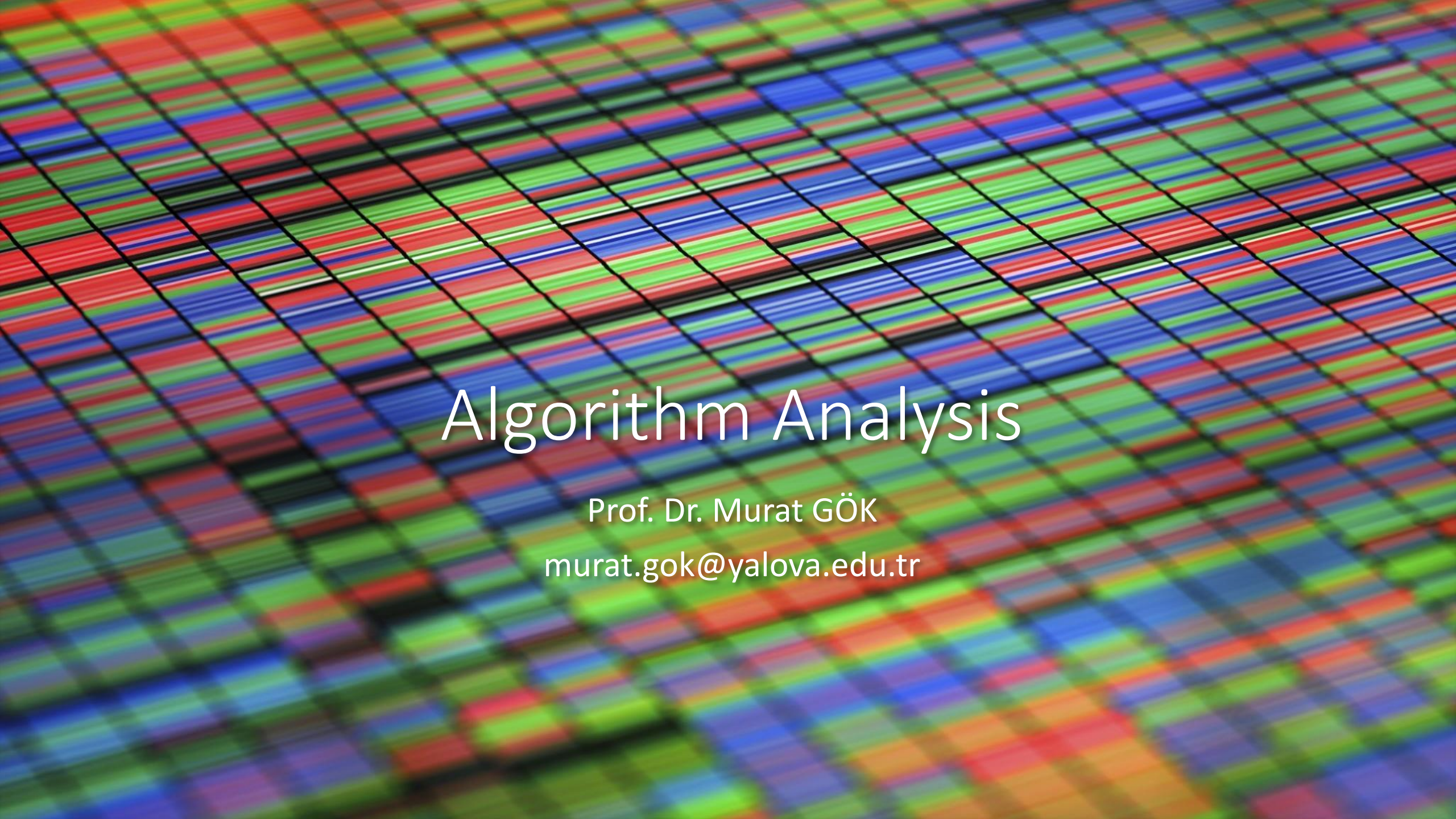


Algorithm Analysis

Prof. Dr. Murat GÖK

murat.gok@yalova.edu.tr

Basics

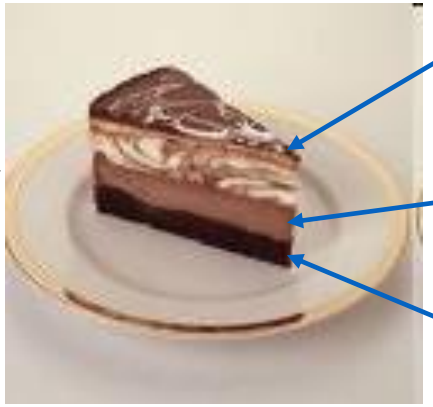
- Variables
  - $x^2+3y-4=1$
- Data Types
  - integer, floating point, unit number, character, string, etc.
  - System-defined data types (Primitive data types)
    - Data types that are defined by system (*int, float, char, double, bool,* etc.).
    - The number of bits allocated for each primitive data type depends on the programming languages, the compiler and the operating system.
  - User-defined data types
    - Data types that are defined by users (*structures* in C/C + + and *classes* in Java and Python).

Data Structures

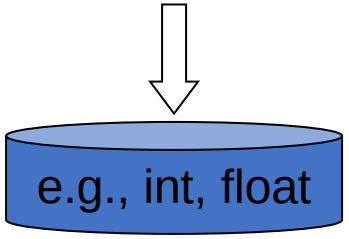
- A data structure is a way of organizing data in a computer so that it can be used efficiently (arrays, linked lists, stacks, queues, trees and graphs).
- Depending on the organization of the elements, data structures are classified into two types:
  - Linear data structures: Elements are accessed in a sequential order, but it is not compulsory to store all elements sequentially (Linked Lists, Stacks and Queues).
  - Non – linear data structures: Elements of this data structure are stored/accessed in a non-linear order (Trees and graphs).

Abstract Data Types

- Abstract data types (ADTs) are used to describe the properties of data types and the operations that can be performed on them.
- We combine the data structures with their operations, and we call this ADTs.
- It is called an abstract data structure because the inside of the structure is completely abstract (does not need to be known) for the user.
- Commonly used ADTs include Linked Lists, Stacks, Queues, Priority Queues, Binary Trees, Dictionaries, Hash Tables, Graphs, and many others.
- Codes are easy to understand using ADTs.
- ADTs provide modularity. A desired ADT variable can be modified without affecting the whole program.
- ADTs can be used later for other programs.



ADTs



Primitive Data Types

What is an Algorithm?

- An algorithm is the step-by-step unambiguous instructions to solve a given problem. Let us consider the problem of preparing an omelette:

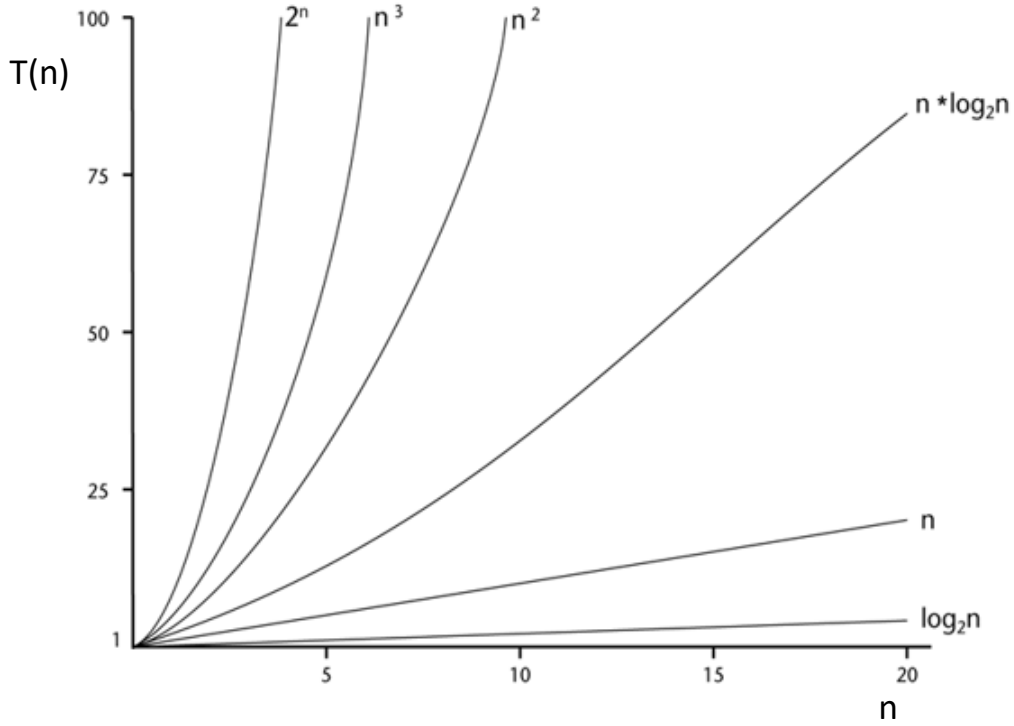
1. *Get the frying pan.*
2. *Get the oil.*
   a. *Do we have oil?*
      i. *If yes, put it in the pan.*
      ii. *If no, do we want to buy oil?*
         1. *If yes, then go out and buy.*
         2. *If no, we can terminate.*
3. *Turn on the stove, etc...*

- Why is the Analysis of Algorithms?
  - Algorithm analysis helps us to determine which algorithm is most efficient in terms of time and space consumed.
- Running Time Analysis
  - The process of determining how processing time increases as the size of the problem (input size) increases.
  - The following are the common types of inputs:
    - Size of an array
    - Polynomial degree
    - Number of elements in a matrix
    - Number of bits in the binary representation of the input
    - Vertices and edges in a graph.

- How to Compare Algorithms:
- To compare algorithms, let us define a *few objective measures:*
  - Execution times?
    - *Not a good measure* as execution times are specific to a particular computer.
  - Number of statements executed?
    - *Not a good measure*, since the number of statements varies with the programming language as well as the style of the individual programmer.
  - Ideal solution?
    - Let us assume that we express the running time of a given algorithm as a function of the input size n (i.e., $f(n)$) and compare these different functions corresponding to running times. This kind of comparison is independent of machine time, programming style, etc.

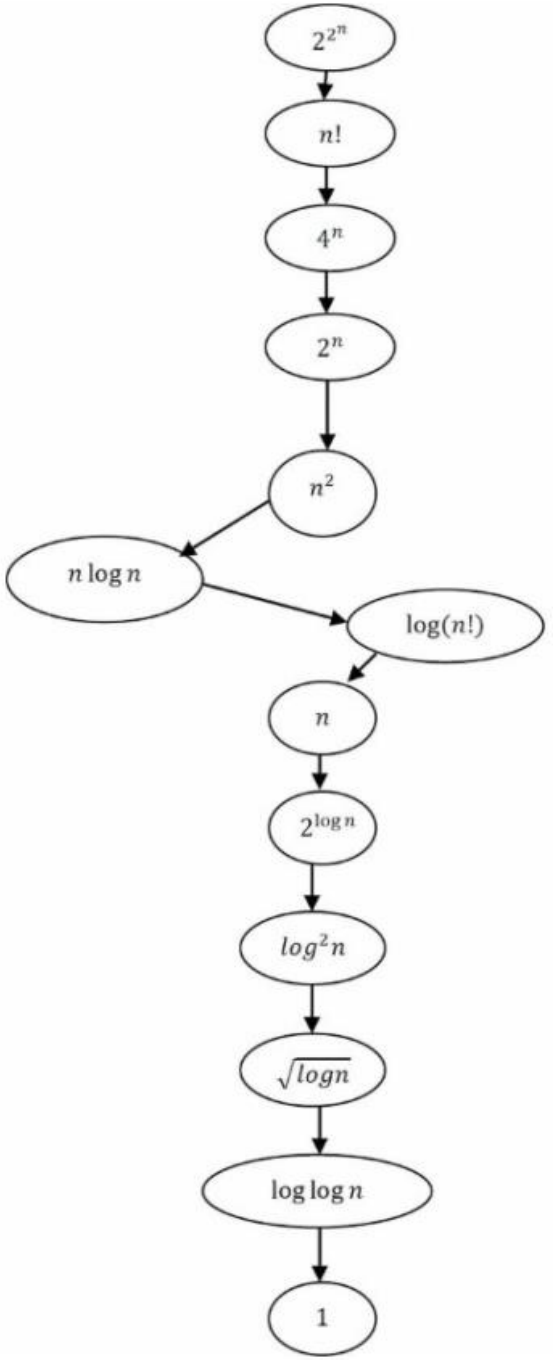
Rates of Growth

- The rate at which the running time increases as a function of input is called rates of growth or growth rate.
- Below is the list of commonly used growth rates:
  - ✓ **O(1)** Time requirement is **constant**, and it is independent of the problem's size.
  - ✓ **O($\log_2 n$)** Time requirement for a **logarithmic** algorithm increases slowly as the problem size increases.
  - ✓ **O(n)** Time requirement for a **linear** algorithm increases directly with the size of the problem.
  - ✓ **O(n*$\log_2 n$)** Time requirement for a **n*$\log_2 n$** algorithm increases more rapidly than a linear algorithm.
  - ✓ **O(n^2)** Time requirement for a **quadratic** algorithm increases rapidly with the size of the problem.
  - ✓ **O(n^3)** Time requirement for a **cubic** algorithm increases more rapidly with the size of the problem than the time requirement for a quadratic algorithm.
  - ✓ **O(2^n)** As the size of the problem increases, the time requirement for **exponential** algorithm increases too rapidly to be practical.



Comparison of Rates of Growth Functions

Function	n = 10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$



DECREASING RATES OF GROWTH

Example

- If an algorithm takes 1 second to run with the problem size 8, what is the time requirement (approximately) for that algorithm with the problem size 16?
  - ✓ If its order is:
  - ✓ **O(1)** ➔ T(n) = 1 second
  - ✓ **O($\log_2 n$)** ➔ T(n) = (1*$\log_2 16$) / $\log_2 8$ = 4/3 seconds
  - ✓ **O(n)** ➔ T(n) = (1*16) / 8 = 2 seconds
  - ✓ **O(n*$\log_2 n$)** ➔ T(n) = (1*16*$\log_2 16$) / 8*$\log_2 8$ = 8/3 seconds
  - ✓ **O(n^2)** ➔ T(n) = (1*16^2) / 8^2 = 4 seconds
  - ✓ **O(n^3)** ➔ T(n) = (1*16^3) / 8^3 = 8 seconds
  - ✓ **O(2^n)** ➔ T(n) = (1*2^{16}) / 2^8 = 2^8 seconds = 256 seconds

Properties of Rates of Growth Functions

1. *We can ignore low-order terms in an algorithm's growth-rate function.*
   - If an algorithm is $O(n^3+4n^2+3n)$, it is also $O(n^3)$.
   - We only use the higher-order term as algorithm's growth-rate function.
2. *We can ignore a multiplicative constant in the higher-order term of an algorithm's growth-rate function.*
   - If an algorithm is $O(5n^3)$, it is also $O(n^3)$.
3. *$O(f(n)) + O(g(n)) = O(f(n)+g(n))$*
   - We can combine growth-rate functions.
   - If an algorithm is $O(n^3) + O(4n^2)$, it is also $O(n^3 +4n^2)$ ➔ So, it is $O(n^3)$.
   - Similar rules hold for multiplication.

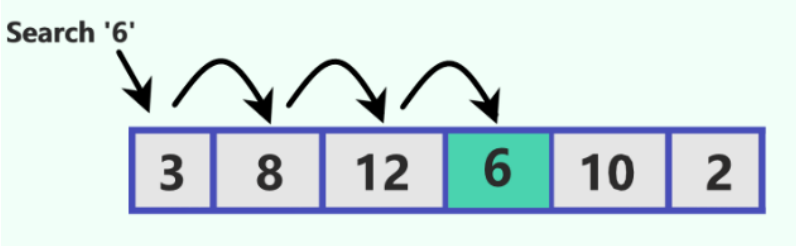
Types of Analysis

- To analyze the given algorithm, we need to know with which inputs the algorithm takes less time (performing well) and with which inputs the algorithm takes a long time.
- In general, the first case is called the best case, and the second case is called the worst case for the algorithm.
- To analyze an algorithm, we need some kind of syntax, and that forms the base for asymptotic analysis/notation. There are three types of analysis:

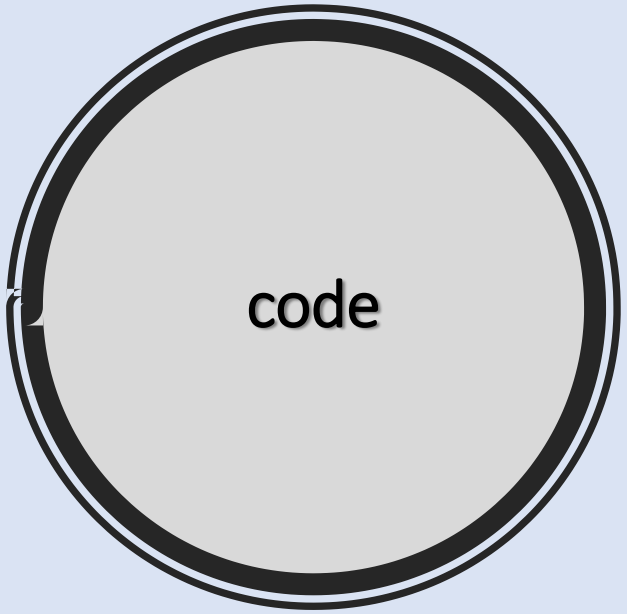
1. **Worst case**
   - Defines the input for which the algorithm takes a long time (slowest time to complete).
   - Input is the one for which the algorithm runs the slowest.
2. **Best case**
   - Defines the input for which the algorithm takes the least time (fastest time to complete).
   - Input is the one for which the algorithm runs the fastest.
3. **Average case**
   - Provides a prediction about the running time of the algorithm.
   - Run the algorithm many times, using many different inputs that come from some distribution that generates these inputs, compute the total running time (by adding the individual times), and divide by the number of trials.
   - Assumes that the input is random.

Example

- Linear search is an algorithm for searching for a specific element in a list. It works by comparing the element to be searched for to each element in the list, one by one, until it finds a match or reaches the end of the list.



- Best case: The best case for linear search is when the element to be searched for is the first element in the list. In this case, the algorithm will only need to compare the element to be searched for to one element in the list and will immediately find a match.
- Worst case: The worst case for linear search is when the element to be searched for is not in the list or is the last element in the list. In this case, the algorithm will need to compare the element to be searched for to every element in the list before it can determine that the element is not in the list.
- Average case: The average case for linear search is when the element to be searched for is somewhere in the middle of the list. In this case, the algorithm will need to compare the element to be searched for to about half of the elements in the list before it can find a match or determine that the element is not in the list.



Python

```python
def linear_search(list, element):
  for i in range(len(list)):
    if list[i] == element:
      return i
  return -1

list = [1, 3, 5, 7, 9]

# Best case
element = 1
result = linear_search(list, element)
print(result)  # 0

# Worst case
element = 11
result = linear_search(list, element)
print(result)  # -1

# Average case
element = 5
result = linear_search(list, element)
print(result)  # 2
```

C++

```cpp
#include <vector>

using namespace std;

int linear_search(vector<int>& list, int element) {
  for (int i = 0; i < list.size(); i++) {
    if (list[i] == element) {
      return i;
    }
  }
  return -1;
}

int main() {
  vector<int> list = {1, 3, 5, 7, 9};

  // Best case
  int element = 1;
  int result = linear_search(list, element);
  cout << result << endl;  // 0

  // Worst case
  element = 11;
  result = linear_search(list, element);
  cout << result << endl;  // -1

  // Average case
  element = 5;
  result = linear_search(list, element);
  cout << result << endl;  // 2

  return 0;
}
```

Asymptotic notation

- Asymptotic notation is a mathematical notation used to describe the running time of an algorithm as the input size grows. It is used in data structures to analyze the performance of different data structures and algorithms.
- The three most common asymptotic notations are:
  - **Big O notation**: Big O notation describes the upper bound of the running time of an algorithm. It specifies the worst-case running time of an algorithm.
  - **Big Omega notation**: Big Omega notation describes the lower bound of the running time of an algorithm. It specifies the best-case running time of an algorithm.
  - **Big Theta notation**: Big Theta notation describes the tight bound of the running time of an algorithm. It specifies the average-case running time of an algorithm.


Thank you for
your attention

Struct (C++)

```cpp
#include <iostream>
#include <string>

// Define a struct named "Person"
struct Person {
    std::string name;
    int age;
};

int main() {
    // Create an instance of the "Person" struct
    Person person1;

    // Assign values to the struct members
    person1.name = "Alice";
    person1.age = 30;

    // Access and print the values
    std::cout << "Name: " << person1.name << std::endl;
    std::cout << "Age: " << person1.age << std::endl;

    return 0;
}
```